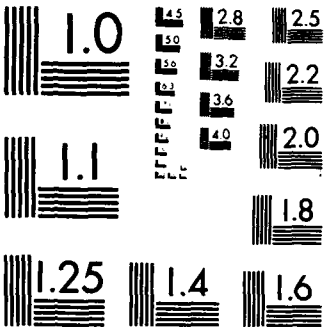
AD-A185 476 OBJECTS ARCHITECTURE: A COMPREHENSIVE DESIGN APPROACH 1/1
FOR REAL-TIME DISTR (U) MARYLAND UNIV COLLEGE PARK
DEPT OF COMPUTER SCIENCE S LEVI ET AL. SEP 87
UNCLASSIFIED CS-TR-1915 N00014-87-K-0124 F/G 12/7 NL



(12)

CS-TR-1915 September 1987

Objects Architecture: A Comprehensive Design Approach for Real-Time, Distributed, Fault-Tolerant, Reactive Operating Systems

Sha[illegible]

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

(12)

CS-TR-1915 September 1987

Objects Architecture: * A Comprehensive Design Approach for Real-Time, Distributed, Fault-Tolerant, Reactive Operating Systems

Shem-Tov Levi and Ashok K. Agrawala

Department of Computer Science
University of Maryland
College Park MD 20742

(*) This research is supported in part by a contract N00014-87-K-0124 from The Office of Naval Research to The Department of Computer Science, University of Maryland.

Objects Architecture: A Comprehensive Design Approach for Real-Time, Distributed, Fault-Tolerant Reactive Operating Systems *

Shem-Tov Levi and Ashok K. Agrawala
University of Maryland
Dept. of Computer Science
Systems Design and Analysis Group
College Park, MD 20742

Abstract

The applicability of objects architecture for designing real-time reactive operating systems is examined through a model of its elements, their relationships and operations. The requirements of creation, deletion and manipulation of objects in a distributed system with a high degree of fault tolerance are introduced and analysed. Then, the imposition of time constraints is introduced, and its influence on the objects architecture is analysed. Relationships with other objects and additional fault tolerance relations are considered. Finally, examples of some solutions to fundamental real-time problems are proposed.

*This work is supported in part by contract N00014-87-K-0124 from The Office of Naval Research to The Department of Computer Science, University of Maryland.

Contents

1 Introduction

An object is a distinct and selectively accessible software element that resides on one of the storage resources of the system. The objects architecture defines the objects as the elements that constitute the system. It also defines their classification, the relationships between them, the set of operations they are subjected to and execution parameters that permit scheduling them for execution and access.

The objects architectures we find today tend to deal with object creation, deletion, access and protection in ways that often become cumbersome for real-time applications. The major inadequacies we find in these architectures are the uncertainties they introduce in access time to objects during execution. For a real time system that manages its resources according to deadlines it has to meet, these inadequacies become serious shortcomings.

We propose an architecture in which object manipulation (creation deletion, access and protection) is handled in a way in which access time can meet the real-time constraints. This proposed architecture further demonstrates time determinism in execution of objects as well, a property that enables predictability of its behavior.

Two major questions are to be answered before considering the ways in which objects are manipulated. The first is the mechanisms of creation and deletion of objects in a distributed system. The second is the way in which objects are accessed, considering requirements for unique identification and requirements for protection imposition. These questions will be discussed in details in the following sections. In this paper we distinguish two classes of objects: executable objects and nonexecutable objects. All the objects are assumed to be addressable by the machines that are used in the distributed system, and representation problems are assumed to be treated locally.

2 Creation and Deletion of Objects

In order to derive the justification for the existence of an object, let us first consider the existential justification of a logical assertion in a database. Such justification takes the form of data dependency mechanism. The existence of an assertion sometimes depends on the existence of one or more other assertions, hence producing a strong data dependency such that modifications (as deletion) must consider all such relations. The justificand is pointed to by a justification which exists as long as its justifiers support it. One should be cautious and verify that a directed cycle does not exist, where the justificand becomes its own justifier. To understand this relationship consider Figure 1. Let one JUSTIFIER be the assertion "$p \Rightarrow q$", and let the second JUSTIFIER be the assertion "p". A trivial JUSTIFICATION can be inferred, and the JUSTIFICAND is then the assertion "q". The

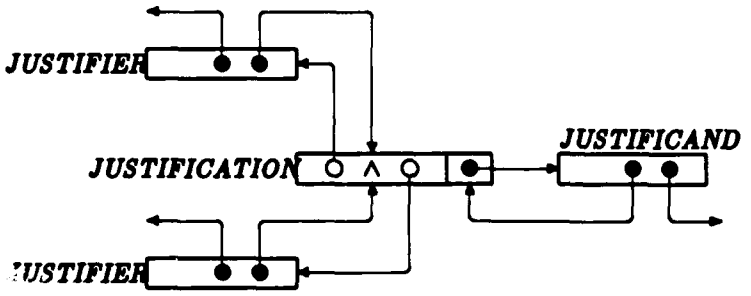


Figure 1: Justifier – Justification – Justificand Relations

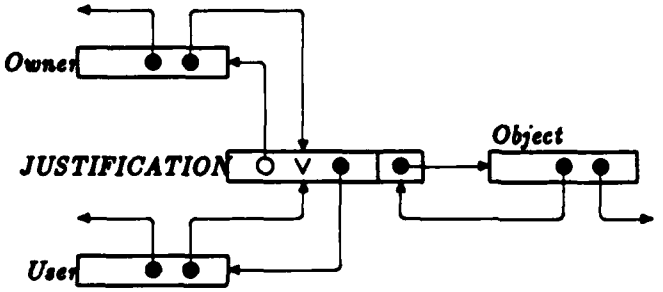


Figure 2: Object's Owner and User Justification

above JUSTIFICATION requires the existance of two JUSTIFIERs, and when one of them is deleted, the JUSTIFICAND should be deleted too. The doubly-linked relationship is very convenient for a search procedure that is expected to move within the network of assertions to manage the data dependencies. For example, it is sufficient for a garbage collection process to examine the JUSTIFICATION in order to mark an object.

When we derive a similar mechanism for the existential justification of an object in a distributed system, we find two types of JUSTIFIERs: a user of the object and the owner of an object. In order to delete such an object, its justification should be unlinked from its owner, and its justification should also have no users linked to it. The reason can be seen in Figure 2. We can then state the following. When an owner of an object creates it, it is the only justifier for its existance. When another user shares the object with its owner, the owner cannot delete it until the user completes the usage. A user that is not the owner of an object can never delete it, unless the owner does. This relationship is trivially expanded to one owner and multiple sharing users. As will be emphasized in the following sections, the JUSTIFICATION can be used to implement mutual exclusion, authorization control and many other tasks.

3 Accessing Objects

For an object to be accessible, the object must be properly identified and the access should be properly authorized. We distinguish the identification and the authorization since identification is for reaching purposes, while authorization also depends on the type of manipulation requirement (as in read permission and write denial).

3.1 Identifying Objects

In order to identify an object, one is required to have the ability of distinguishing it from the set of accessible objects. This ability is based on a property of uniquely naming each of the accessible objects at each level of the system operation. The term "name" therefore encounters more than just an appellative nature. It may be better interpreted as a precise pointer to the object.

Each name is always interpreted with respect to a particular *context* ([16]). The context is a set of bindings of names to objects. We show in the following section how we propose to attach properties to contexts.

In distributed systems we find different types of names at different layers of the system architecture ([2]). Each layer must maintain a uniform view ([20]) at its distributed partitions when referring to an object. Therefore, there is a need to manage mappings for the name spaces used, keeping each name unique for all accessibility levels ([14]):

1. Character *string* names: used in the file system and in user programs.
2. Segment *numbers*: used by a running process to refer to an active segment.
3. A segment table (of a user) provides physical *addresses* for the page table of a corresponding segment.

Context initializing (also called binding, or linking, or loading) is the bridge between the high level human oriented names, to the low level machine oriented addressing world. Furthermore, the need to convert human identifiers to machine identifiers requires the use of some kind of a catalog that maps the two finite spaces. Allowing possible access of multiple users, extends the above requirement to a multiple catalog system. Here each catalog can be viewed as a context in which the identifiers are interpreted. In other words, different catalogs can use the same "name" for different objects, and we can still distinguish between them. The ability to selectively share a context is made possible if catalogs appear as named objects in other catalogs. The network that results is called *naming network* ([16]). When a context is shared, there is a need to distinguish between the name given to an object by its owner, and the name by which an external user refers to this object. A

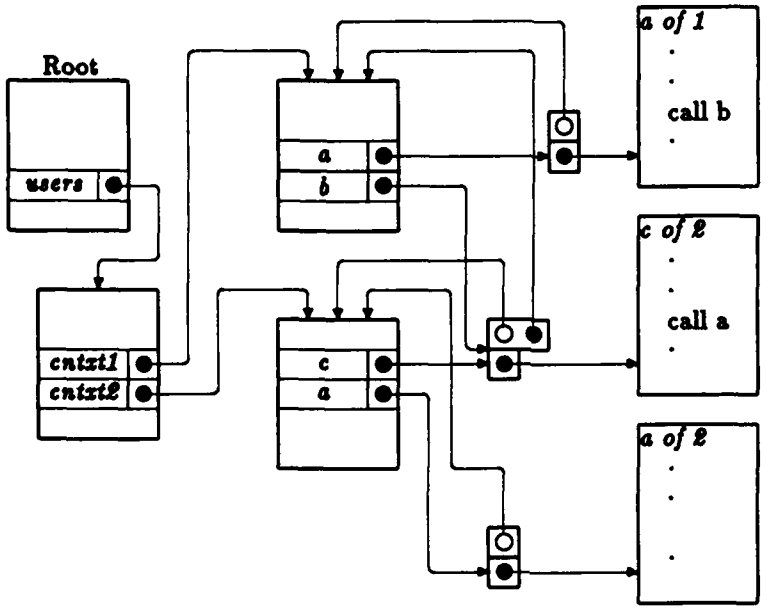


Figure 3: Example: Different Contexts Sharing an Object

trivial solution is to impose uniqueness of names globally, but it is impractical. A better solution is a mechanism that connects an object that contains other objects' names to the proper contexts in which these names are bound. The simplest way to do it is to use the catalogs as such mechanisms. Yet, generally we want to allow the usage of different names at different contexts when referring to the same shared object, and therefore, the catalogs are not sufficient. Hence one needs dedicated pointers constructs (*closures* in [16]) that serve as *joints* between objects accessed and the contexts (catalogs) that attempt to access them.

Recalling the scheme discussed in section 2, the *joint* is the JUSTIFICATION, the catalogs (also called directories) are the JUSTIFIERS, and the accessed object is the JUSTIFICAND. For example consider Figure 3. An object *a* of context1 refers to an object whose name according to this context is *b*. Yet, this object is named *c* according to context2, which owns it. Furthermore, when the latter refers to *a*, it is *a* according to context2. Object *a of 1* is justified by the catalog of context1, while object *a of 2* is justified by the catalog of context2. The object *c of 2* is justified by two catalogs. Its *joint* points forward to *c*, and backward to its two justifiers. The knowledge of "who's the owner?" is also kept within the joints.

The overall scheme is called *the name service*, and is supported by the operating system.

A context initialization procedure is activated when a bridge between different naming levels does not exist. A context initializing procedure usually consists of resolving and installing actions. The resolution of a name involves searching an existing object that is identified by a given name in a given context.

The unknown time characteristics in such a search disqualifies it as a possible run time procedure in a real-time system. In a real-time system the context initializer must be allowed only as an off-line task,[1] probably executed before running a real-time job. In addition, in order to be able to adhere to timing constraints, the response time of a name service should always be reasonably bounded. In order to achieve such bounds name servers might be limited to small-size unique names, in order to reduce search time. Such a restriction raises issues of reusability of names, recalling the need to maintain the uniqueness of names in a given context, especially regarding the selectively shared contexts objective. However, implementing such a service correctly allows any user to disconnect itself from external objects, but requires some additional tools for deletion of objects because other users may be linked to them (see section 2).

The search rules in a name service in a real-time operating system should be efficient from the real-time point of view. As such, a direct entry in a directory should point a joint of an object, and an indirect entry should specify the whole access path to the shared context, to the joint in concern. Applying such an approach allows limiting the number of directories that should be searched for an access (for example, the user's working directory, a language library and a system library). The language and system librarys can be kept ordered and balanced, and thereby reduce search time to a minimal bounded time.

Some examples of distributed operating systems and their name servers are given by Tanenbaum and Von Renesse ([18]), reviewing their distributed properties. The Cambridge operating system has a single name server process that maintains names mapping on line. The V operating system uses a three level mapping context servers. Amoeba ([13]) and Eden ([1]) operating systems use a capability ([5,8]) system naming scheme.

3.2 Protecting Objects

Real-time embedded systems usually neglect protection mechanisms. However, a real-time operating system cannot. Stringent real-time requirements, being the dominant ones, necessitate keeping the time taken by the protection mechanism as low as possible. For example, consider the case in which users are being given a direct access to required resources on remote sites. The access time is reduced, but it is done hand in hand with increasing the availability of a potentially restricted information. Most of today systems

[1] See section 4.1 for the definition of "off-line".

cannot be that limited from a protection point of view.

Protection systems are divided into two categories: list oriented systems and ticket oriented systems (as capabilities [5]). Access control lists imply search procedures that are not adequate for real-time systems. Capability system, in which authorization is a prior phase to run time, provides a better real-time environment.

The *joint* introduced above is suitable for maintaining a ticket oriented protection scheme. Its advantage is that its authorization test is carried out prior to run time, before binding it to the user's context. Furthermore, a user needs a backward link from the joint in order to gain access to an object, and thus the joint acts as an information hiding module. Embeding an access control process in it may support mutual exclusion control as well.

In a distributed system, an application program does not have an explicit knowledge about storage locations it uses. Therefore a service which maintains a link to remote sites must be given. Furthermore, timing properties of different allocation instances may differ significantly. Therefore, the efficiency of such a service is very significant.

Some examples of protection mechanisms and servers are given in [18]. Amoeba ([13]) and Eden ([1]) operating systems use capabilities as a ticket oriented protection mechanism. Yet, in the Amoeba operating system rights are protected cryptographically, while in the Eden operating system the kernel supports the protection. The Cambridge operating system uses a list oriented protection scheme. The Eden operating system presents an interesting approach in its file service. Each object has its private file server as a process embeded in it, and the directory service is separate. This principle may be applied to the protection mechanism of the joints, as mentioned above.

4 Time Constraints and Objects

After considering a sample of timing problems in accessing and protecting objects, a more detailed examination of the timing constraints is appropriate. But definitions of time constrains require some prior definitions for other time notions.

Let $C_i(t)$ be the monotonic function that maps real-time (global time) to clock-i time. Each computation is assumed to have accessibility to some clock in the system. The clocks in the system are *inaccurate* due to a nonzero drift-rate. This inaccuracy results in *incorrect* time readouts, in other words $C_i(t) \neq t$. Clock synchronization algorithms ([9,11,19,6]) result in having a local drift rate which is bounded. Therefore, each computation is assumed to be able to access a clock, and acquire the estimate of the current real-time $C_i(t) = t \pm \Delta_i(t)$.

We define *event* as a detectable instantaneous atomic change in a system state. In a

real-time system, the state of the system includes the set of clocks $\{C_i(t)\}$. Let $C_i(t) = C_0$ be the set of system states in which the readout of clock-i is C_0. Hence we can define the following predicates on system states.

$$Taft(C_0)$$

which is *true* for $C_i(t) \geq C_0$ and *false* otherwise.

$$Tbef(C_d)$$

which is *true* for $C_i(t) \leq C_d$ and *false* otherwise.

4.1 Time Constraints

Informally speaking, a time constraint is a requirement to start executing a particular executable object, after a condition is satisfied, and complete the execution before a deadline is passed. The execution time of the object is assumed to be given, and the constraint is extended to a periodic execution of the object.

A time constraint is formally defined as

$$< Id, Taft(condition_1), c_{Id}, f_{Id}, Tbef(condition_2) >$$

where:

Id is the name of the executable object (process) in the proper context in concern,

$Taft(condition_1)$ states after what event should execution begin. Simple *true* stands for "as soon as possible",

c_{Id} is the computation time of object Id,

f_{Id} is the frequency in which the computation should be carried out in case this is a periodic process. In case of a spontaneous (sporadic) process, this is the maximal frequency expected. $f_{Id} = 0$ stands for a single occurence of execution,

$Tbef(condition_2)$ states the deadline $d_{Id} = C_d - Now$ which should be met. $condition_2 \equiv C_{Id}(t) = C_d$ with $C_d = \infty$ stands for an "off line" computation.

The "window" defined from $Taft(condition_1)$ to $Tbef(condition_2)$ delimits the domain in which the executable object is allowed to execute.

4.2 Real-Time Reactive Operating System and Time Constraints

A real-time reactive operating system uses the time constraint as the key to its decisions on execution initiation and resource scheduling. A reactive system repeatedly responds to requests from its environment by producing outputs ([7]). In our case the requests are invocations of executable objects. While in regular operating systems the scheduling mechanism is entirely application independent, and is maintained as an internal issue of the operating system, in real-time operating systems the resource management and allocation mechanism should adhere to application constraints. This strong relationship between a real-time operating system and the application programs implies that when a real-time application job is running, the system's global discipline state is relatively constant. Yet, there are scheduling disciplines that a real-time system can never support. One example is a round robin discipline, in which the overhead of switching control between processes does not adhere to timing efficiency. The most general scheduling is divided to two parts:

- *Off-line* scheduling: a process in which the management of the policy of a discipline of a service scheduling is dynamically updated. For example, when a resource is added to the system, in addition to recognizing the availability of that resource a new discipline may be adequate.
- *On-line* scheduling: application of current discipline.

Scheduling can be considered as maintaining ordered lists (queues), whose items are members of a given set, and the items are sorted according to a specified key. In our model the given set is the set of real-time constraints. Two important properties of the above model hold in case a working solution is feasible.

1. $\forall i$ in the model: $c_i < d_i < 1/f_i$.
2. $\sum_i c_i f_i \leq$ the number of available resources.

The first property must hold continuously, and is a consideration of the "on-line" scheduler, while the latter is a condition for the "off-line" scheduler. The latter requirement is stronger than what is really needed: it is based on the maximal demand rate the system can find due to spontaneous processes.

From the above model we can derive the property that distinguishes a real-time operating system from others:

- In a real-time operating system the scheduling decisions are based on the real-time constraints (the above quintuple) of the active processes in the system.

A variety of scheduling disciplines of real-time constraints and their properties is discussed by A. Mok in [12], and we do not deal with it in this paper. One aspect we do emphasize is that the on-line scheduler requires the knowledge of the time constraints imposed on an object in order to schedule it. Therefore, this property should be attached to the object, and allow its efficient scheduling.

The on-line scheduler is activated by process requests and by time servers. Its implementation should be very efficient, at "kernel" level, so that the overhead of management is minimized. The off-line scheduler can be implemented at a user process level and not necessarily in the kernel level. User processes are also allowed to be off-line, e.g. for log-in, compile, bind and so on. As such, they can be served in any time independent discipline (LIFO, FIFO etc.), a policy which is used for ordering processes that have the same non-critical "latency".

The time constraint contains a parameter c_{Id} which is assumed to be given and to characterize the executable object's timing characteristics. We have shown above the importance of this parameter. Now we show how this parameter is affected by the objects architecture of the system. Furthermore, we state necessary properties this parameter has to describe.

4.3 Time Constraints Characteristics of an Object

A model of the execution time of a time constraint can be constructed (e.g. [10]) of five components:

1. Execution time of the task on the processor (E_i) which depends on the task size (T_i) and the processor MIPs rate (μ)[2]:

$$E_i = T_i/\mu.$$

2. The time required to access other objects when executing this object (Ac).

3. The communication network and operating system overhead (Ov), which is used for concurrency control, integrity checking, recovery check-point update etc.

4. Inter-processor communication (IPC), whose cost is higher if communicants reside on different processors.

5. Waiting time (WT) which is consumed when the task waits in the processor enablement queue. This figure depends highly on the sizes and number of tasks, the

[2]Better estimators, but more complicated ones, have been derived. For examples, a schedulability analyzer for Real-Time Euclid ([17]) is known to provide estimates of execution time bounds with an error smaller than 25%. Since it is not the scope of this paper, we have chosen to refer to this simple model.

processor load, and the number of enablements (especially if large tasks are assigned to the same processor). This figure is the main reason for disqualifying round robin discipline.

Therefore, the total execution time of a process can be constructed

$$E_T = \sum_i (E_i) + Ac + Ov + IPC + WT.$$

Enhencement of the performance of a system, as well as increasing the system's margin away from being unable to satisfy the demands, is done by minimizing E_T. For a given network, Ov and the number of enablements is relatively a constant. Hence in order to reduce E_T the following steps should be adopted:

- Reduce WT: large tasks should be assigned to different processors.
- Reduce IPC: tasks with high IPC cost with each other, should be assigned to the same processor.
- Reduce E_i: large tasks should be assigned to processors with higher MIPs rate.
- Enhance Ac: reduce access time to objects, using short path to point them, as the *joints* above.

The ways in which reduction of IPC and WT is achieved are not within the scope of this paper.

When considering the above, one can clearly state that for any decision making by a scheduler, information about the timing properties of an executable object must be attached to the object, in addition to protection characteristics and others. We propose the *joint* as the construct in which properties are attached and according to which decisions regarding access are taken. This policy is uniform for both executable and nonexecutable objects. In order to enhance E_T, the context initialization procedure must be divided into two parts: an off line *binder*, and an on line *loader*. The binder executes all the authorization tests and "connects" the joints of accessed objects to the authorized user object prior to execution of the object. Then, during run-time, the loader uses direct links to manipulate the accessed objects. Not only that access times are reduced, but by using this methodology they are also deterministic and predictable.

4.4 Executable and Non-Executable Object Joints

So far, we defined a *joint* to contain the following parts:

- A context independent pointer for the naming network, to allow a multi-user selective sharing of the object.
- An owner/user justification list.
- A ticket check mechanism for the protection scheme.

In addition, for an executable object the time constraints of the object should be within the joint to allow scheduling it. A time constraint

$$< Id, Taft(condition_1), c_{Id}, f_{Id}, Tbef(condition_2) >$$

can be expanded further. There might be more than one possible configurations to execute this object. In such a case c_{Id} is replaced by a list $< c_{Id}[k] >$, which reflects the computation time of the possible configurations. Furthermore, Id may be a construct of some processes that are related to each other in a specific way. In such a case Id is replaced by a graph, say P, of time constraints, which is compatible with the communication graph of the system, say C, such that the edges of P represent the specific relations.

The distinction between executable and non-executable objects is, therefore, more than in the executability property. In a real-time system the time constraints also play an important role in this distinction. An executable object is a time constraint on the system resources, while the non-executable is not. Furthermore, even an off-line job has a time constraint, but its deadline is simply infinity. Therefore, these two object classes differ not only in their internal view, but also in their external view, or in other words in the joints that point to them.

Practical aspects may shed some light on this distinction. Consider a single processing node that contains a processor, a stable memory (disk, tape, etc.), and a volatile memory. This processing node is an instance of a portion of some processing site, allocated for an execution of a particular job. The process to be executed may reside both on the volatile memory and on the stable memory. This process is accessable by the *on-line scheduler* and it has a time constraint based on the processing node allocated to execute it. It is an executable object, pointed to by a joint that contains a proper time constraint. Now consider an object that contains the source code of this process in its text representation. This object is a data for some *compiler* and is not executable. Even the binary representation of this process, the output of the compiler, is not an executable object. It serves as data for the *binder*, even though some parts of it may be an exact replica of some parts of the process. The binder is the machine that adds to the object's joint properties that concern its relationships to other objects, as well as its time constraints, and thereby produce an executable object. The *loader* and the *off-line scheduler* might modify the constraints (e.g. replacing a relative temporal expression by an absolute one, replacing a list of possible

execution configurations by the allocated ones). The visibility of an object to other objects is restricted both by the protection mechanism and by its type. For example, an execution of the source code of a process (a non-executable object) is rejected by its type, without the need to check access authorization.

5 Relationships Between Objects

Objects have been extensively considered in software engineering, dealing with an establishment of a software development technique. In this context an object is captured as an entity whose behavior is characterized by the operations it is subjected to and the operations it carries out on other objects. The external view of an object (these operations) is its specification, and the internal view of an object is its implementation. In this paper our point of view is slightly different. We consider the use of objects architecture in a system context, thereby expanding the above object definitions to describe elements and entities in a more general way. Yet, some of the properties that characterize objects in software development context ([3]) are valid in the system architecture context as well:

- An object has a state.
- There is a set of actions to whom an object is subjected and a set of actions it requires from other objects.
- It is denoted by a name.
- It has a restricted visibility of (as well as by) other objects.

5.1 Relationships and Operations

In our model, objects that relate to each other, or in other words objects that satisfy a given semantic link, are connected via the owner/user justifications in the joint. These relationships are in accordance with the visibility restrictions and the set of operations to whom the justificand is subjected and the justifier operates on. We can model a system as a graph whose nodes are objects and whose arcs are directed from justifier to justificand representing the owner/user relationship.

Executable objects can be divided ([3]) into three type classes, depending on their relationships with other objects. Since our definition of an object is slightly different from that of software engineering, we modify this classification:

- *Actor*: An object which is subjected to no operation. It only operates on other objects.

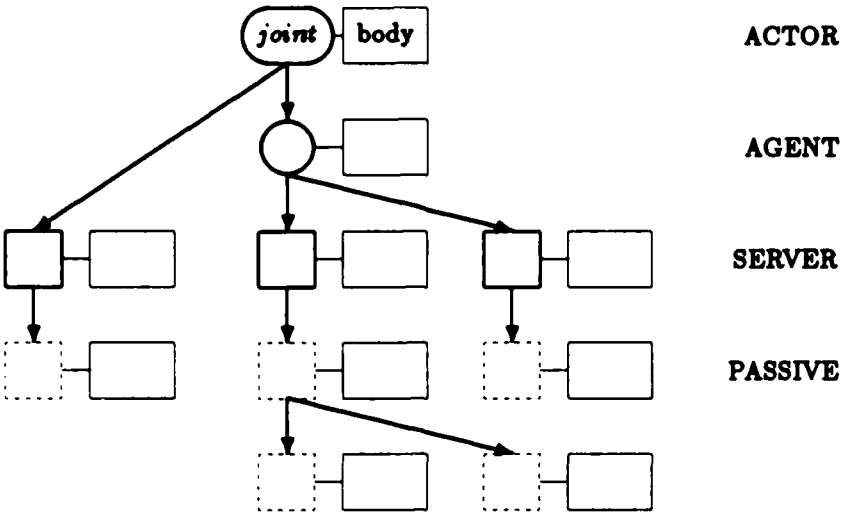


Figure 4: Relations between different types of objects.

- *Server*: An object which is only subjected to operations by other executable objects. It does not operate on other executable objects, but it may operate on non-executable objects.
- *Agent*: An object which operates on (one or more) executable objects on behalf of another executable object. In turn other executable objects may operate on it.

In addition there are the non-executable objects, which are only subjected to operations.

- *Passive*: An object which is only subjected to operations and does not operate on others.

An example of such a system can be seen in Figure 4. It is important to notice that the owner/user relationship is not an operation, although when an executable object is involved it is in accordance with a set of permittable operations. For example consider the relationships between passive objects. Since a passive object does not operate on other objects, its relationships with other objects are mainly of an existential justification nature, as in fault tolerant copies or in a distributed database. This type of relationships necessitate the grouping of objects of the same type into a meta-object, to whom the rest of the objects may refer to as a whole entity. We differ the grouping of a set of executable objects into a *troupe* ([25]) from the grouping of a set of non-executable objects into a *pack*. The dissimilarity originates in the different requirements of fault tolerance, which are discussed in the next section.

The set of operations which an object is allowed to be subjected to, as well as the set of operations which can be carried out by an object, are in accordance with the object type. For example, a passive object has an empty set of operations it is allowed to carry out on executable objects. G. Booch ([3]) defines three classes of operation types on objects:

- Operations that change an object state.
- Operations that evaluate current state of an object.
- Operations that allow visiting parts of an object.

Recall that these operations can be carried out on object bodies as well as on object joints.

5.2 Fault Tolerance Relations

Requirements for fault tolerance impose maintenance of redundant objects, with topology that adhere to the fault tolerance specifications. In an object oriented system, a support for maintenance of objects that are copies or versions of other objects is required. It is the operating system that should support this maintenance, according to the application specification, but without the application's continuous control. There might be any whole number of copies, or versions, to an object, according to the requirements of the system to whom the original object belongs. We distinguish between copies or versions of an executable object, denoted as *alternatives*, and copies of a non-executable object, denoted as *replicas*. Both alternatives and replicas raise an additional constraint that must be satisfied, the *consistency* constraint. This constraint adds a new dimension to our model of a system, through semantic links between the original object and its alternatives or replicas.

Fault tolerance of executable objects utilizes mechanisms which allow recovery from a fault. The most common mechanisms used in order to recover a system upon detection of an error are roll-back mechanisms ([36]). In real-time systems a simple roll-back strategy might lead to a deadline miss, which is an unrecoverable fault itself. A deadline miss might occur if an error is detected when there is not sufficient time to activate an alternative. Hence, in real-time systems alternatives should be activated soon enough, even before their original executable object (the preferred alternative) begins an acceptance test. Such a roll-forward approach is usually used in a modular redundancy design and in N-Version programming (NVP). Its major drawback is the high degree of *determinism* it requires. A set of alternatives is completely deterministic if each alternative has exactly one possible execution for each of its reachable states. A set of alternatives of an original executable object (a troupe [25]) may decrease the degree of determinism required (and increase asynchrony) by allowing more application dependency ([25]) or by forcing all the

troupe members to satisfy a set of conditions ([35]). In cases of executable objects that apply roll-forward approach, the consistency constraint is raised mainly by contentions. In the roll-back mechanisms, such contention may be solved by the use of an audit trail, created by a coordinator object for the update of semi-idle cohort objects which serve as backup ([22]). This checkpoint mechanism supports the maintenance of the availability of accessable executable objects for a specific job, while maintaining information on required start-up points for the backup alternatives. Problems of consistency of objects that are concurrently executing are usually solved ([32]) by means of a very expensive atomic broadcast ([38,23,26]), which is unsuitable for real-time applications. Its inappropriateness is due to the stringent timing constraints to which the communication network is subjected. An atomic broadcast requirement might therefore overload the communication network, or at least reduce its efficiency, to an undesired level. Hence, the above solution is impractical. Another extensively researched area in checkpointing is determining the optimal interval for checkpointing ([24,28,29,30]). However, in checkpoint interval research, real-time constraints have not been taken into account.

Fault tolerance of non-executable objects is mainly achieved by means of maintaining replicas of the objects and enforcing a uniform serializability of the operations on all replicas of each object. The property of ensuring that concurrent execution of operations on a passive object and its replicas is equivalent to a serial execution on the passive object is called *one-copy serializability*. A method which is very common in ensuring that replicas are consistent is a weighted voting. An example can be found in [31], for replicas of a passive object which is subjected only to read and write operations. In this example, consistency is maintained by setting a read quorum and a write quorum such that non-null intersection between any read quorum and any write quorum is ensured. This method is highly suitable for the roll-forward schemes of the executable objects, although it is serial to a concurrent execution of the alternatives. Its determinism is very high, especially from execution time aspects. Maintaining consistency is also achievable via atomic actions ([21,34,37]). For a distributed (message passing) system, commit protocols ([39,40,33]) provide the means to implement the atomic actions. The success of recovery procedures depends on satisfaction of recoverability conditions ([41]) by the object's internal view (its implementation). Atomic actions are very expensive for the reasons previously stated. Namely, application of this approach in a real-time system consumes an enormous amount of resources.

The use of replicas and alternatives raises issues of partitioning of a pack or a troupe, and particularly the issue of how to ensure a one-copy serializability of replicas across partitions. A survey of both syntactic and semantic strategies that deal with the above issues is given in [27]. In cases where a transaction is not allowed to be rolled-back for a recovery procedure (thereby preempting execution), a *pessimistic* approach must be

adopted. This approach prevents inconsistency by limiting availability, assuming the worst on the other partitions. On the other extreme, if the availability is more important, or if an access to a part of a pack is sufficient, an *optimistic* approach may be adequate. This approach allows global inconsistency to occur and later (upon connection of the partitions) be recovered by some resolution mechanism, as undoing, compensating or correcting. The choice between the two approaches is application dependent, however it seems that the more restricted a system is, the more pessimistic its approach is going to be.

6 Examples in Real-Time Design Issues

6.1 Example 1: Interrupt Driven Systems

Interrupt driven systems are frequently used in real-time applications, especially in control systems. Each interrupt that triggers the system is responded by an interrupt service process, as defined by the system designer. This service process is invoked by the operating system, through its interrupt handler. The handler identifies the arriving interrupt, and passes control to the adequate service process accordingly.

An interrupt service is defined as a time constraint too. The designer of the system defines the maximal frequency in which each interrupt is allowed to occur. A deadline for the service of each interrupt is distinctly set, along with computation requirements. Conditions on start time and termination time (deadline) are also defined by system designer. Hence, each of the interrupt service processes is expressable as a time constraint, using the same quintuple defined in section 4.1. However, an additional mechanism is required to ensure that a particular service process is to be executed if and only if the interrupt which it serves has already arrived. In other words, a time constraint of an interrupt service process depends on the occurrence of an *event*, and not only on the start condition.

The requirement to guarantee deadlines in a real-time system implies that the scheduling mechanism must take interrupts into account. Most proposed scheduling mechanisms ignore the effect of services given to interrupts. The simplest example of a faulty approach is the one which allows an interrupt to be served the moment it occurs, preempting the object currently being scheduled to execute. In such a preemption, one might have generated a deadline miss, which might be an unrecoverable fault. Furthermore, there might be enough time to execute the service process after the currently executing process completes, without any preemption at all. The above reasons justify a selective preemption approach, and the objects architecture we propose is fully suitable for it. As we have mentioned above (section 5), an object has a *state* that can be evaluated and can be changed. This property may be used to deal with interrupt driven processes in a way which does not contradict a guaranteed deadline.

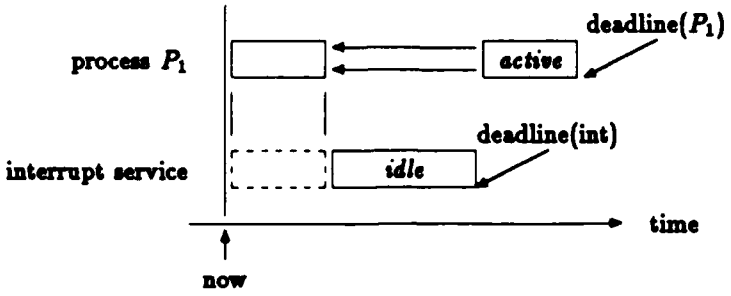


Figure 5: An interrupt service without preemption

6.1.1 Interrupt Driven Objects

Let an executable object, which is activated as a response to an interrupt, have two states: *idle* and *active*. Let the state of such an object be *idle* as long as the interrupt it serves has not occurred. The off-line scheduler, the binder and the loader, which participate in the allocation of resources, must take the interrupt service requirements into account. There is no other way which guarantees the deadline of this service upon interrupt occurrence, while maintaining the guarantees for the deadlines of the already *active* objects. Therefore, the *idle* and the *active* objects are all queued, regardless what their states are. The on-line scheduler applies a *state-evaluation* procedure to the queued object which is to use the resource next. If it is an *active* object, then this object receives control on the resource. If it is an *idle* object, its execution is postponed. If there exists an *active* object, that is scheduled for a later time and whose execution can be carried out ahead of time (without violating the postponed object's time constraint), then this *active* object receives control on the resource. Such a case is described in Figure 5. P_1 can be carried out before serving the interrupt. There is no reason to preempt P_1, even if the interrupt occurs during the execution of P_1.

An *idle* object turns to be an *active* one by the *state-change* procedure, which is invoked by the interrupt handler according to the interrupt source. Both the *state-evaluation* procedure and the *state-change* procedure are very short (regarding their execution time) and consume a constant overhead cost for all the devices at a specific processing node. This results from the direct access to objects through the *joint*'s justification, as described in section 2.

However, applying a very safe approach as in Figure 5, without allowing a violation of the time constraint of the postponed interrupt server, is wasteful. Hence, we want to utilize the slack[3] of an *idle* object, as long as it is *idle* and its time constraint is not con-

[3]Time slack is defined ([12]) as $\max(d_{Id} - c_{Id}, 0)$ where d_{Id} is the time to deadline and c_{Id} is the computation time.

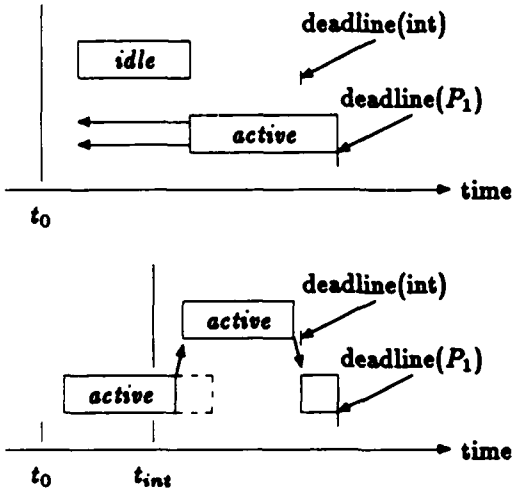


Figure 6: "First-Deadline" scheduling with preemption

flicted. Such an example is given in Figure 6. P_1 executes although the interrupt service deadline is earlier, until the service time constraint is conflicted. At the conflict detection the service preempts P_1. One of the problems with this approach is the overhead cost of the context switching. An additional problem is the depth of preemption allowed, while guaranteeing all constraints to be satisfied. These problems are solved in our comprehensive methodology, by selectively allowing such switches and only at checkpoints. Thus, preemption is allowed only at a finite number of points in the execution trace of an object.

6.2 Example 2: Communication Service as an Agent Object

Communication between two distinct processing nodes serves the following goals:

1. processes that execute on different nodes make a synchronization attempt, or
2. a process on one node initiates an object migration to the other, or
3. an object is in need for a remote service.

In the synchronization case, real-time systems differ from others. A communication synchronization attempt, which is not bounded with a timeout check, might introduce a deadline miss. Therefore, an unbounded communication synchronization violates an already given execution time guarantee, or even prevents by its existence any possibility for a guarantee at all. However, there are other solutions to synchronization. Examples are

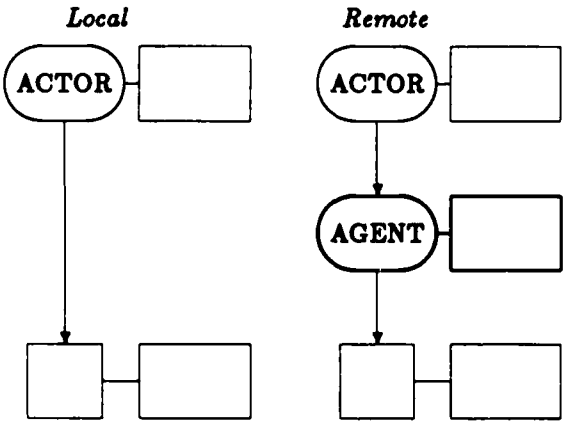


Figure 7: Local vs remote servers

provided through clock synchronization and calendars management for resource allocation and scheduling. The migration case is not considered as an on-line task. The size of an object might be modified in run-time, a fact that prevents a possible guarantee for a migration execution time. Therefore, we do not consider these two possible communication occurrences (synchronization and migration) in this paper.

However, a remote service is very frequent in distributed systems. A request of a remote service is due to a lack of an adequate *sever* locally. It may originate in executive and functional reasons, as well as in fault tolerance ones. The guaranteeing procedure that is used for local computations is not adequate in such a case. One needs some *agent* process to interface between the local *actor*[4] and the *remote server*. This agent process must acquire all the knowledge on execution time parameters by the server. Then, the agent must impose a real-time constraint on the proper server, such that the actor satisfies its own real-time constraint. The difference between a local and a remote service is described in Figure 7.

6.2.1 Remote Service Considerations

Analysis of the tasks such an agent has to perform, results in a set of activities which should take place in both the actor's node and the server's node, as demonstrated in Figure 8. Agent a (at node A) cannot guarantee that the execution of the service b (at node B) will take only t_0 time units, which is the time required to execute b at B. Additional time is needed due to:

1. communication delays t_{comm},

[4]The request for a remote server may be initiated by an agent as well. We use an actor in our example just for simplicity of the description.

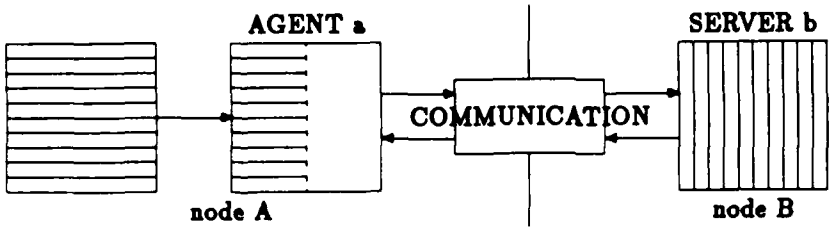


Figure 8: Objects involved in communication

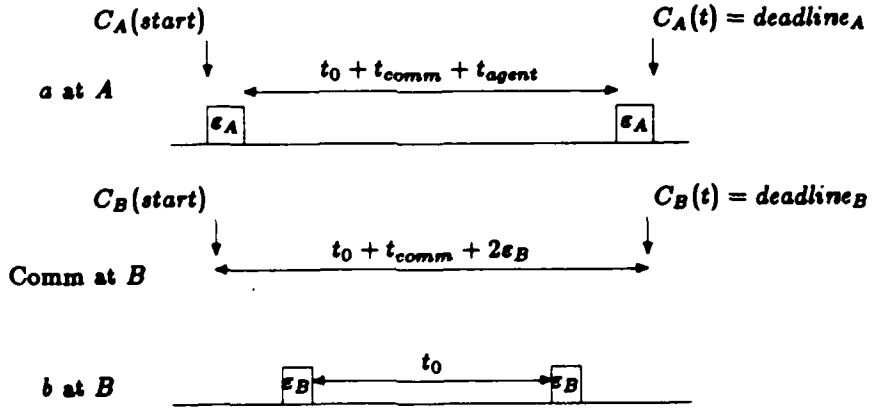


Figure 9: Timing diagram for local and remote nodes

2. clock inaccuracies ε_A and ε_B,

3. agent overhead t_{agent}.

These timing considerations are described in Figure 9. Server b at B can guarantee an execution interval of $t_0 + 2\varepsilon_B$. The communication portion at B extends this interval to $t_0 + t_{comm} + 2\varepsilon_B$. This results in an execution interval $t_0 + t_{comm} + t_{agent} + 2\varepsilon_A$ at node A. If node A and node B are homogeneous, then a local execution in A requires only t_0.

In this model, the communication is therefore a part of the guaranteed execution time. This property is effective not only during the execution phase, but also during the allocation and scheduling phases. The communication is not regarded as a point-to-point message delivery, but rather as an agent which is responsible to satisfy guaranteed deadlines at remote nodes.

7 Concluding Remarks

Object-oriented system design seems to provide means to construct systems with a high degree of deterministic and predictable timing properties. This determinism, together with the required fault tolerance schemes, result in a time constraint oriented system. In this paper we summarized our current point of view on object types, the set of operations each of the object types is associated with and their relationships. A conceptual model of creation, deletion and access for manipulation and state verification that we have considered in our analysis of the applicability of objects architecture for a real-time distributed fault tolerant operating system, lead us to define the *joint* that contains the following parts:

- A context independent pointer for the naming network, to allow a multi-user selective sharing of the object.
- An owner/user justification list.
- A ticket check mechanism for the protection scheme.
- A time constraint for an executable object.
- A replica/alternative control mechanism for the fault tolerance scheme.

We discussed the distinction between on-line and off-line (infinite deadline) execution of objects. Scheduling executable objects and context initialization are each divided in our model into an on-line part and an off-line part. The context initializer consists of an off-line *binder* and an on-line *loader*. Scheduling policy and acceptance of jobs (requests to execute an object) are managed by the *off-line scheduler* which allocates the resources

before loading. The *on-line scheduler* carries out locally the policy, and dispatches loaded objects according to their time constraint.

In accordance with the above, we gave two examples of solutions to problems from the real-time domain in which our objects architecture is used. We have shown that the interrupt service can be given preemptively or non-preemptively without conflicting the alredy given guarantees. In addition, a different approach considering the communication system for remote object invocations has been introduced, and some of the considerations needed for a concurrent execution have been presented.

We intend to extend this research and consider in details a model of an executable object, and a non-trivial time constraint, to allow alloction, scheduling and verification of a distributed real-time computation. Since we apply a top down approach in our system requirements, we will analyze the effects of concurrency on time-services as well.

References

[1] Almes G. T., Black A. P., Lasowska E. D. and Noe J. D., *The Eden System: A Technical Review*, IEEE Trans. on Software Engineering, Vol SE-11 No 1 pp 43-59, Jan 1985.

[2] Bochmann G., *Distributed Systems Design*, Springer-Verlag, Berlin Germany, 1983.

[3] Booch G., *Object-Oriented Development*, IEEE Trans. on Software Engineering, Vol SE-12 No 2 pp 211-221, Feb 1986.

[4] Caspi P., Halbwachs N., *A Functional Model for Describing and Reasoning Time Behavior of Computer Systems*, Acta Informatica, Vol 22 No 6 pp 595-628, March 1986.

[5] Farby P. S., *Capability Based Adressing*, Communication of the ACM, Vol 17 No 7 pp 403-412, July 1974.

[6] Gora W., Herzog U. and Tripathi S., *Clock Synchronization on the Factory Floor*, Proc. of Workshop on Factory Communication, NBS, March 1987.

[7] Harel D. and Pnueli A., *On The Development of Reactive Sysytems*, Weitzman Institute of Science, Rehovot, Israel, 1985.

[8] Kain G. Y., *On Access Checking in Capability Based Systems*, IEEE Trans. on Software Engineering, Vol SE-13 No 2 pp 202-207, Feb 1987.

[9] Lamport L. and Melliar-Smith P. M., *Synchronizing Clocks in the Presence of Faults*, Journal of the ACM, Vol 32 No 1 pp 52-78, January 1985.

[10] Ma P., Lee E., Tsuchiya M., *Design of Task Allocation Scheme for Time Critical Applications*, IEEE Proceedings - Real Time Systems Symposium, Miami Beach FA, Dec 1981.

[11] Marzullo K. and Owicki S., *Maintaining the Time in a Distributed System*, ACM Operating Systems Review, Vol 19 No 3 pp 44-54, July 1985.

[12] Mok A., *Fundamental Design Problems for the Hard Real Time Environment*, MIT Ph.D. Dissertation, Cambridge MA, May 1983.

[13] Mullender S. J. and Tanenbaum A. S., *The Design of a Capability Based Operating System*, The Computer Journal, Vol 29 No 4 pp 289-299, Aug 1986.

[14] Popek G., *Issues in Kernel Design*, (in *Operating Systems: An Advanced Course*, Bayer R., Graham R. and Seegmuller G. – editors), Springer - Verlag, Berlin, Germany, 1979.

[15] Shankar A. U. and Lam S. S., *Time-Dependent Distributed Sytems: Proving Safety, Liveness and Real-Time Properties*, technical report CS–TR–1586, Department of Computer Science, University of Maryland, College Park MD, Dec 1985.

[16] Saltzer J., *Naming and Binding of Objects*, (in *Operating Systems: An Advanced Course*, Bayer R., Graham R. and Seegmuller G. – editors), Springer - Verlag, Berlin, Germany, 1979.

[17] Stoyenko A. D., *A Case for Schedulability Analyzable Real-Time Language*, IEEE Fourth Workshop on Real-Time Operating Systems, Cambridge MA, July 1987.

[18] Tanenbaum A., Von Renesse R., *Distributed Operating Systems*, ACM Computing Surveys, Vol 17 No 4 pp 419-470, Dec 1985.

[19] Tripathi S. and Chang S., *A Clock Synchronization Algorithm for Hierarchical LANs – Implementation and Measurements*, Technical Report TR-86-48, Systems Research Center, University of Maryland, College Park MD, 1986.

[20] Watson R. W., *Distributed System Architecture Model*, (in Lampson B. W., Paul M., Siegert H. J. (editors), *Distributed Systems: Architecture and Implementation – an advanced course*, pp 10-43), Springer Verlag, Berlin 1981.

[21] Allchin J. E. and McKendry M. S., *Synchronization and Recovery of Actions*, 2'nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Aug 17-19, 1983.

[22] Birman K. P., Joseph T. A., Raeuchle T. and El-Abbadi A., *Implementing Fault Tolerant Distributed Objects*, IEEE Trans. on Software Engineering, Vol SE-11 No 6 pp 502-508, June 1985.

[23] Birman K.P. and Joseph T. A., *Reliable Communication in an Unreliable Environment*, TR85-694, Department of Computer Science, Cornell University, Ithaca NY, July 1985.

[24] Chandy K. M., Browne J. C., Dissly C. W. and Uhrig W. R., *Analytic Models for Rollback and Recovery Strategies in Data Base Systems*, IEEE Trans. on Software Engineering, Vol SE-1 No 1 pp 100-110, Mar 1975.

[25] Cooper E. C., *Replicated Procedure Call*, ACM Operating Systems Review, Vol 20 No 1 pp 44-55, Jan 1986.

[26] Cristian F., Aghili H., Strong R. and Dolev D., *Atomic Broadcast: from Simple Message Diffusion to Byzantine Agreement*, IEEE Fifteenth Fault Tolerance Computing Symposium (FTCS), pp 200-206, 1985.

[27] Davidson S. B., Gracia-Molina H. and Skeen D., *Consistency in Partitioned Networks*, Computing Surveys, Vol 17 No 3 pp 341-370, Sept 1985.

[28] Gelenbe E. and Derochette D., *Performance of Rollback Recovery Systems under Intermittant Failures*, Communication of the ACM, Vol 21 No 6 pp 493-499, June 1978.

[29] Gelenbe E., *On The Optimum Checkpoint Interval*, ACM Journal, Vol 26 No 2 pp 259-270, Apr 1979.

[30] Gelenbe E., Finkel D. and Tripathi S. K., *Availability of a Distributed Computer System with Failures*, Acta Informatica, Vol 23 pp 643-655, 1986.

[31] Gifford D. K., *Weighted Voting for Replicated Data*, ACM Operating Systems Review, Vol 13 No 5 pp 150-162, Dec 1979.

[32] Joseph T. A. and Birman K. P., *Low Cost Management of Replicated Data in Fault Tolerant Distributed Systems*, ACM Trans. on Computer Systems, Vol 4 No 1 pp 54-70, Feb 1986.

[33] Lakshman T. V. and Agrawala A. K., *Efficient Decentralized Consensus Protocols*, IEEE Trans. on Software Engineering, Vol SE-12 No 5 pp 600-607, May 1986.

[34] Liskov B. and Scheifler R., *Guardians and Actions: Linguistic Support for Robust Distributed Programs*, ACM Trans. on Programming Languages and Systems, Vol 5 No 3 pp 381-404, July 1983.

[35] Mancini L., *Modular Redundancy in a Message Passing System*, IEEE Trans. on Software Engineering, Vol SE-12 No 1 pp 79-86, Jan 1986.

[36] Randel B., *System Structure for Software Fault Tolerance*, IEEE Trans. on Software Engineering, Vol SE-1 No 2 pp 220-232, June 1975.

[37] Reed D. P., *Implementing Atomic Actions on Decentralized Data*, ACM Trans. on Computer Systems, Vol 1 No 1 pp 3-23, Feb 1983.

[38] Schneider F. B., Gries D. and Schlichting R. D., *Fault Tolerant Broadcasts*, Science of Computer Programming, Vol 4 pp 1-15, North Holland, 1984.

[39] Skeen D., *Nonblocking Commit Protocols*, Memorandum No UCB/ERL M81/11, Electronic Research Lab, College of Engineering, University of CA, Berkeley, March 1981.

[40] Skeen D. and Stonebraker M., *A Formal Model of Crash Recovery in a Distributed System*, IEEE Trans. on Software Engineering, Vol SE-9 No 3 pp 219-228, May 1983.

[41] Taylor D. J. and Seger C. J., *Robust Storage Structures for Crash Recovery*, IEEE Trans. on Computers, Vol C-35 No 4 pp 288-295, Apr 1986.

8c. ADDRESS (City, State, and ZIP Code)

10 SOURCE OF FUNDING NUMBERS			
PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)

Objects Architecture: A Comprehensive Design Approach for Real-Time, Distributed, Fault-Tolerant, Reactive Operating Systems.

12. PERSONAL AUTHOR(S)

Shem-Tov Levi and Ashok K. Agrawala

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	September 1987	26

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

END

DATE

FILMED

DEC.

1987